Competing in Correlation One's Terminal with Reinforcement Learning

Neo Jun Hao (A0183783R), Tran Minh Duong (A0184528W), Lim Feng Yue (A0210754M)
CS4246 Team 14

1. Introduction

Terminal is a 2-player Tower Defense game with simultaneous turns organized by Correlation One. They hold frequent competitions which allow university students to compete for a prize (Correlation One, 2021). The game is fully deterministic, with access to perfect information at all times. With 420 positions in the game, 6 types of units, and health information of each unit, taking a naive approach to state space would result in a state size of (4 x health) ^ 420. Also, a total combination of 1134 actions can be taken, and given that we can take multiple actions in a single turn, this makes the action space exponentially huge. In addition, most player-submitted algorithms on Terminal are hard-coded and utilise little to no machine learning aspects (Terminal Forum).

Some reinforcement learning techniques are known to perform poorly with large state and action spaces. We also considered techniques such Monte Carlo Tree Search and Adversarial Search but deemed them unsuitable due to the large state and action space, which would elongate the decision time and result in a penalty. Hence, our group decided to train a one-size-fits-all Deep Q Network (DQN), and evaluate the ability and flexibility of the algorithm to play against different strategies. We divided the labour of attack and defence into separate agents and reduced the state and action space to a linear size. We also tested out different variations of the state space, Q-network structure, and training conditions to find the best performing agent. Due to the nature of the competition, we are unable to obtain the source code of top-ranking algorithms from other players and decided to use the given algorithm from the template code, Python-algo, to train our DQN. We observe that our DQN agents generally perform better than the baseline Python-algo. Our top performing agent, NoBS_Copy_100_2, had an outstanding performance against agents with different strategies and came in the top 1% of algorithms (algos) for the Terminal Season 8 competition.

2. What is Terminal?

Terminal is a 2-player Tower Defense game with simultaneous turns on a 28 x 28 diamond battlefield. One player takes the top triangular half of the battlefield, while the other player takes the bottom field. The winning condition of the game is to reduce your opponent's health to zero before they do the same to yours. You can do so by deploying **Mobile units** to attack your opponent or setting up **Structure units** to defend against your opponent's Mobile units. These can be deployed by using Mobile points (MP) and Structure points (SP) and cost varies depending on the units.

There are 3 types of Mobile units and 3 types of Structure units. The Mobile units include the Scout which does fast and light damage, the Demolisher which deals damage to opponent Structure units, and the Interceptor which does damage to opponent Mobile units. The Structure units include the Wall, a physical obstacle that influences pathing, the Support that shields nearby friendly Structure units, and the Turret which damages opponent Mobile units. Units are deployed within the player's half of the battlefield, Mobile units can only be deployed at the edge, and Structure units can be deployed anywhere. The opponent is damaged if your Mobile units reach the edge of the opponent's playing field and likewise for us. Apart from deployable units, there is also an Upgrade and Remove action, which upgrades existing Structure units to increase their stats or remove them for a partial refund on points.

In Terminal, a turn occurs in 3 phases. The first is the Restore phase, where both players are given Mobile and Structure points to deploy units. This is followed by the Deploy phase, where each player chooses the units to deploy, at which location, and can deploy multiple units. This ends off with the Action phase, where all units are deployed and the game engine plays out the outcome. When either player's health points (HP) reaches zero, the game ends and the player with the higher HP wins. There is a limit of 100 turns per game, at the end of which the player with the higher HP wins. Further details of game mechanics and game rules can be obtained from the website.

3. Agent architectures

There are three different agent architectures that we've constructed. For each agent architecture, there are two DQNs: one is in charge of defense actions while the other is in charge of attack actions. This allows us to significantly reduce action space and learning time as each agent has its own Q-network and replay buffer that is catered for its functionality. These agent architectures share the same action space, reward function, and learning approach, and the only differences among the different agent architectures are their state space and the use of a feed-forward neural network or convolutional neural network for the DQN.

3.1. General agent architecture

3.1.1. Action space and action generation procedure

The action space for all three agent architectures is the same. In detail, the defense agent can perform 210 x 4 + 1 = 841 actions. There are 210 locations on our half of the arena and each location allows for deployment of 1 of the 3 Structure unit types or upgrading an existing Structure unit. An additional END action is also included for the agent to stop deploying. Likewise, the attack agent follows a similar structure. It has 28 x 3 + 1 = 85 actions as there are 28 edge locations on our side and 3 types of mobile units to deploy and the END action.

Our agents will learn and generate actions in the deploy phase. The process of generating actions in the deploy phase is also the same for all agent architectures. First, we let the defense agent apply defense actions until we run out of Structure points, reach a limit of 10 actions or get an END action. Then, we let the attack agent similarly deploy Mobile units with Mobile points. The reason for this ordering is because the layout of the Structure units will affect the pathing of Mobile units, thus we choose to deploy Structure units first. Throughout the process, the resulting state after an action will be the input state for generating the next action.

However, the defense agent faces additional constraints which the attack agent does not have. The highest Q-value action for a state might be an illegal action due to the rules of the game: we cannot deploy multiple Structure units at one location, upgrade a location with no Structure units, or deploy Mobile units at locations that already have Structure units. To overcome this, we utilise a modified version of the epsilon-greedy approach by only choosing executable actions. The action-choice process is shown in **Figure 1**.

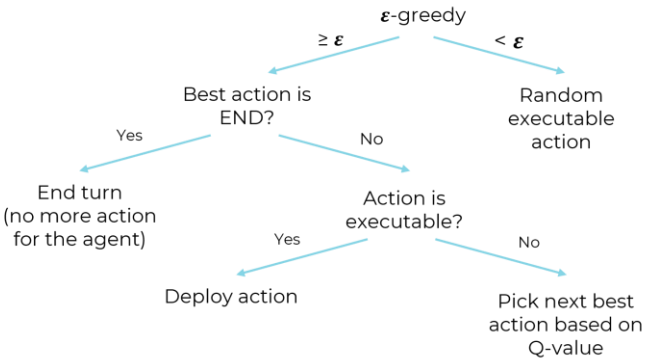

Figure 1. Modified ε-greedy algorithm for choosing an action on a given state

3.1.2. Reward function

The reward function is also the same for all agent architectures. It expresses heavy penalties for losing HP, big rewards for chipping down the enemy's HP, small rewards for bringing down the enemy's Structure units, and small penalties for losing our Structure units. With how the Deploy phase works, it is difficult to define the exact reward for each action. Hence, we assign the same reward value to all actions within the same Deploy phase. The reward is calculated as the difference between a function of the previous state and a function of the current state, and its calculation is shown in **Figure 2**.

$$R(s, a, s') = \begin{cases} r(s') - r(s), & \text{if non-terminal} \\ 10000, & \text{if win} \\ -10000, & \text{if lose} \end{cases}$$

$$r(s) = \alpha * (my\_health - enemy\_health) + \beta * [(my\_structure\_points + my\_structure\_units) - (enemy\_structure\_points + enemy\_structure\_units)]$$

Where $\alpha = 10$, $\beta = 1$

Figure 2. Reward function for all model architectures

3.1.3. Learning approach

At the start of a turn, the agent stores experiences from the previous turn as tuples of (state, next state, action, reward, done) in the replay buffer, and the neural network is fitted by temporal difference (TD) learning. TD learning is performed on a stochastic batch of 64 experiences. The replay buffer holds up to 10000 experiences for the agents with a feed-forward neural network and 1000 experiences for agents with a convolutional neural network.

3.2. Feed-forward DQN agent

The state space for this agent is encoded as a 1D vector of 425 values. The first 5 values are HP, enemy HP, turn counter, mobile points, and structure points while the other 420 values correspond to every coordinate on the gameboard and their corresponding Structure unit if any: -1 - empty, 0 - WALL, 1 - SUPPORT, 2 - TURRET, 3 - SCOUT, 4 - DEMOLISHER, 5 - INTERCEPTOR. The state space for this agent does not contain the breached and scored locations (**Figure 3**). **Figure 4** shows the feed-forward neural network architecture, with an input layer of 425 nodes, two hidden layers of 128 nodes, and an output layer of 841 (defence actions) or 85 (attack actions) nodes.

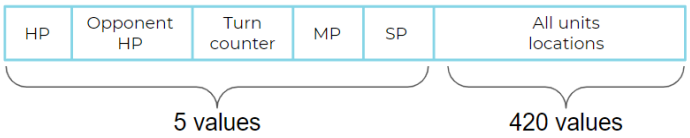

Figure 3: State space for feed-forward DQN agent

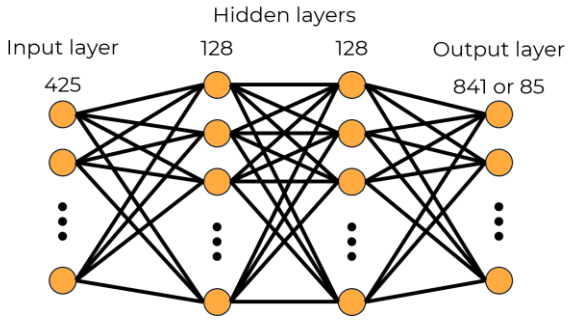

Figure 4. Feed-forward network for the DQN agent

3.3. Feed-forward DQN agent with breached and scored locations.

The state space for this agent is encoded as a 1D vector of 481 values. The first 425 values are the same as the 425 values in **section 3.2**. The next 28 values represent the breached locations, containing every coordinate of the edges for our agent and the damage the enemy did via Mobile units. The last 28 values represent the scored locations, where we scored on the enemy (**Figure 5**). The feed-forward neural network for this agent is the same as for the agent in **section 3.2**, except with an input layer of 481 nodes.

Figure 5. State space for feed-forward DQN agent with breached and scored locations

3.4. Convolutional DQN agent.

The state space for this agent is encoded as a 3D matrix of 28 x 28 x 6 = 4764 values. For each location of the game's 28 x 28 board, we record 6 values, which depends on whether the location is in the arena bound or not. If the location is in the arena bound, the 6 values will be a one-hot encoding of the unit type. Otherwise, the 6 values will be HP, enemy HP, turn counter, MP, SP, and an unutilised constant 0 (**Figure 6**). The convolutional neural network structure is shown in **Figure 7**.

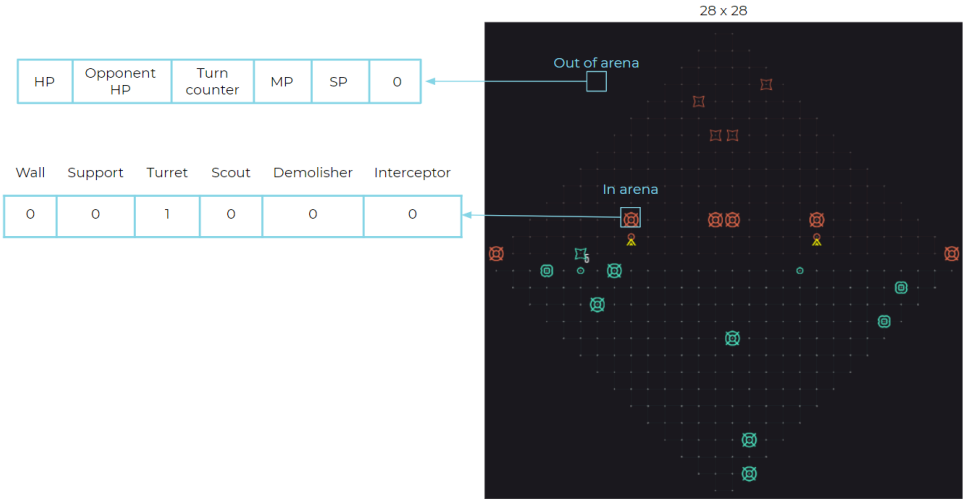

Figure 6. State space for convolutional DQN agent

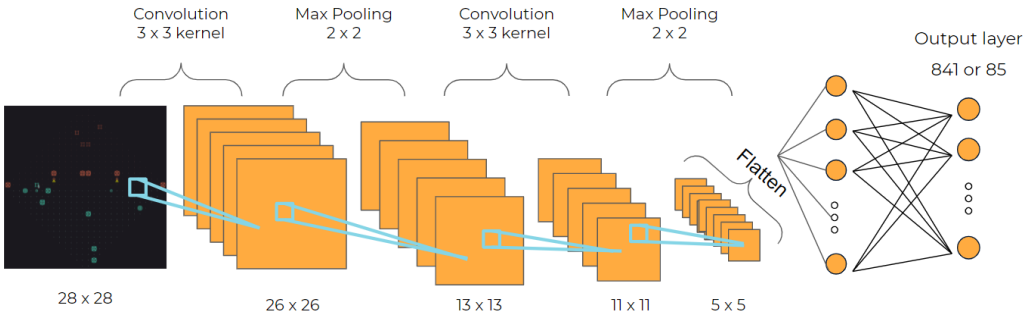

Figure 7. Convolutional neural network for DQN agent

4. Results

We have 3 sets of agents specified in **Section 3**. For each algorithm, training of the DQN was done against the starter algorithm (Python-algo), or against itself where the code is first duplicated then played against each other. Since local access to the source code is required for training, and only Python-algo's source code is available, we are unable to train our agent against a wider variety of algorithms. In terms of the performance of the agent, we observe its rating and win rate against other agents. Rating is obtained from Correlation One's Terminal website after they have played at least 100 games against other Algos. This is to minimise fluctuations in the rating score to give a better indication of their true performance. To test the generalizability of the agent, we played it against Python-algo and 3 test agents provided on Correlation One's Terminal website - Foreman, Infiltrator, and Ironclad - with increasing difficulty respectively according to the website. Foreman has light offence and defence, Infiltrator has heavy offence, and Ironclad has a heavy offence and a good defence.

4.1 Without breached/scored locations

First, we tested our model without including the breached and scored locations of the Mobile units specified under **Section 3.2**. These agents use the feed-forward neural network. For this, we trained 5 different agents on different conditions:

- 100 games against Python-algo - NoBS_Python_100
- 100 games against copy of itself (1) and (2) - NoBS_Copy_100_1 and NoBS_Copy_100_2
- 200 games against copy of itself (1) and (2) - NoBS_Copy_200_1 and NoBS_Copy_200_2

The labels (1) and (2) indicate that they are from the same training batch i.e. (1) is trained on 100 games against (2), and both are DQNs. For ease of mentioning our agents, we use a naming convention as follows: (Agent type)_(Training condition)_(Number of games)_(Label). As an example, '100 games against a copy of itself' is named Copy_100_1, or Copy_100_2. For agents trained against Python_algo, there is only one Deep Q Network, hence its name would be NoBS_Python_100, with no label. For this, "NoBS" indicates the exclusion of breached and scored locations in the state space.

The agent with the highest rating is NoBS_Copy_100_2 with 1587, followed by NoBS_Copy_200_1 with 1574. Smite is the winning Algo of the Summer 2021 competition and has the highest rating of 1900. The baseline Python-algo has a rating of 1366. We observe that all of our agents performed reasonably better than Python-algo in terms of rating. All of our agents defeated Python-algo in all 5 games, except NoBS_Python_100, which lost 1 game. The two agents with the highest ratings also had the highest flexibility, being able to win all five games against Foreman and Ironclad, while winning three out of five games against Infiltrator. There was also no increase in rating as the number of training games increased. The results are compiled in **Table 1**.

	Smite	NoBS_ Python_ 100	NoBS_ Copy_100 _1	NoBS_ Copy_100 _2	NoBS_ Copy_200 _1	NoBS_ Copy_200 _2	Python-algo
Rating	1900	1418	1519	1587	1574	1558	1366
Python- algo	-	4/1	5/0	5/0	5/0	5/0	-
Foreman	-	2/3	5/0	5/0	5/0	5/0	5/0
Infiltrator	-	1/4	2/3	3/2	3/2	1/4	1/4
Ironclad	-	0/5	2/3	5/0	5/0	0/5	0/5

Table 1. Results of agent performance without breached/scored locations

Agents are assessed based on their rating, their performance against the baseline Python-algo, and their performance against different agents: Foreman, Infiltrator, and Ironclad. Performance is shown as a Win/Loss ratio in a series of five games.

Across all agents, we observed a distinct three-corner deployment strategy that is deployed by the agent. This is shown in **Figure 8**, with NoBS_Copy_100_2 as the playing agent. At the start of the game, the agent immediately deploys Structure units at the left and right corners of the gameboard. Over time, as the game progresses, the agent goes on to build up Structures along the centre bottom of the battlefield. However, the agents differ from each other in how many Structure units they dedicate to the centre bottom area. This is most obvious when comparing NoBS_Copy_100_2 against NoBS_Copy_200_1. We observe that NoBS_Copy_100_2 builds up a full column of Structure units, as opposed to NoBS_Copy_200_1, which only builds up half the number of Structure units.

When analysing the attack patterns of the agents, we observed a significant difference between agents trained on Python-algo and those trained against a copy of itself. All agents trained against a copy of itself have a strong preference towards Interceptors, while NoBS_Python_100 has a preference for Demolishers, but still deploys Scouts and Interceptors. All the agents also typically have a favourite deployment location, where it deploys a majority of its Mobile units from, and a small amount in the surrounding area (not shown). This occurs most often near the middle of either the bottom left or bottom right edge. The deployment patterns are shown in **Table 2**.

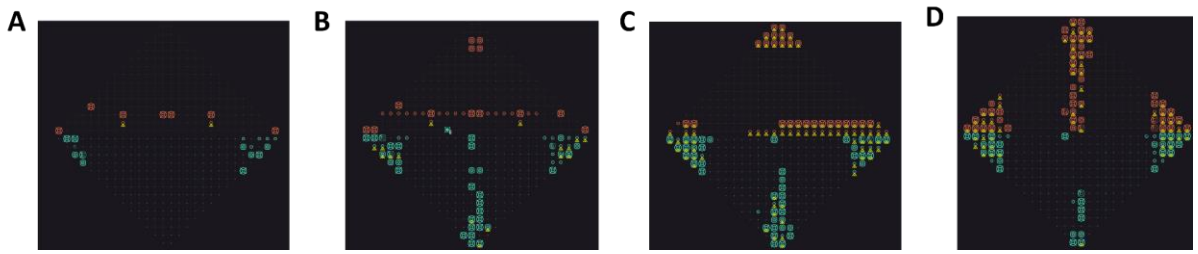

Figure 8. Gameboard of Terminal with agents without breached/scored locations
(A) *NoBS_Copy_100_2 (bottom) VS Python-algo (top) - start of game.*
(B) *NoBS_Copy_100_2 (bottom) VS Python-algo (top) - end of game.*
(C) *NoBS_Copy_100_2 (bottom) VS Ironclad (top) - end of game.*
(D) *NoBS_Copy_100_2 (bottom) VS NoBS_Copy_200_1 (top) - end of game.*

Units/Agent	NoBS_Python_100	NoBS_Copy_100_1	NoBS_Copy_100_2	NoBS_Copy_200_1	NoBS_Copy_200_2
Scout	21.8%	0.4%	0.2%	0.4%	0.7%
Demolisher	**66.9%**	0.0%	0.6%	1.4%	0.7%
Interceptor	11.3%	**99.6%**	**99.2%**	**98.2%**	**98.6%**

Table 2. Mobile unit deployment patterns of agents without breached/scored locations
Values are the percentage of Mobile points spent on the respective Mobile units, obtained from 1 winning game of the agent against Python-algo, where applicable. Scout and Interceptor cost 1 Mobile point, Demolisher costs 3 Mobile points.

Notably, NoBS_Copy_100_2 was our best algo, coming in at **#234** out of 65526 algos submitted by 40304 players for the Season 8 competition (**Figure 9**).

Figure 9. Position of NoBS_Copy_100_2 in the leaderboard
The submitted algo name is NOBREACH2-100-SELF

4.2 With breached/scored locations

Next, we tested our models with the inclusion of breached and scored locations into the state space, while using integer representation to record actions in the replay buffer. These agents also use the feed-forward neural network. "BS" stands for inclusion of breached and scored locations into the state space. The conditions tested are as follows:

- 100, 200, and 300 games against Python-algo - BS_Python_100, BS_Python_200, BS_Python_300
- 100 games against copy of itself (1) and (2) - BS_Copy_100_1 and BS_Copy_100_2
- 200 games against copy of itself (1) and (2) - BS_Copy_200_1 and BS_Copy_200_2
- 300 games against copy of itself (1) and (2) - BS_Copy_300_1 and BS_Copy_300_2

For brevity of the report, agents trained against a copy of itself only have the better of the duo analysed E.g. BS_Copy_100_2 has a much higher rating than BS_Copy_100_1, and so BS_Copy_100_1 is excluded for analysis. With this batch of agents, the agents with the highest ratings were also those with the lowest number of training games i.e. 100 (**Table 3**). We observe a noticeable decrease in rating as the number of training games increases, for both BS_Python and BS_Copy. This trend is also observed in the games against Foreman, Infiltrator, and Ironclad, as the total number of wins decreases consistently over the number of training games. The agents here as a whole performed more poorly than those trained without breached/scored locations.

	Smite	BS_ Python_1 00	BS_ Python_2 00	BS_ Python_3 00	BS_ Copy_ 100_2	BS_ Copy_ 200_1	BS_ Copy_ 300_2	Python-algo
Rating	1900	1456	1279	1111	1452	1306	1107	1366
Python-algo	-	5/0	2/3	0/5	5/0	4/1	2/3	-
Foreman	-	5/0	5/0	5/0	5/0	5/0	5/0	5/0
Infiltrator	-	1/4	0/5	0/5	1/4	0/5	0/5	1/4
Ironclad	-	0/5	1/4	0/5	1/4	0/5	0/5	0/5

Table 3. Results of agent performance with breached/scored locations

Agents are assessed based on their rating, their performance against the baseline Python-algo, and their performance against different agents: Foreman, Infiltrator, and Ironclad. Performance is shown as a Win/Loss ratio in a series of five games.

Contrary to the three-corner strategy observed in agents trained without breached/scored locations, we do not observe that strategy here (**Figure 10**). The deployment pattern of the agents is much more dispersed, with a preference for placing Structure units in the centre to the right-centre area. As the number of training games increases, the agents also begin to deploy fewer Structures. They learn to deploy fewer Structures at the beginning of the game and deploy fewer Structures up until the end of the game.

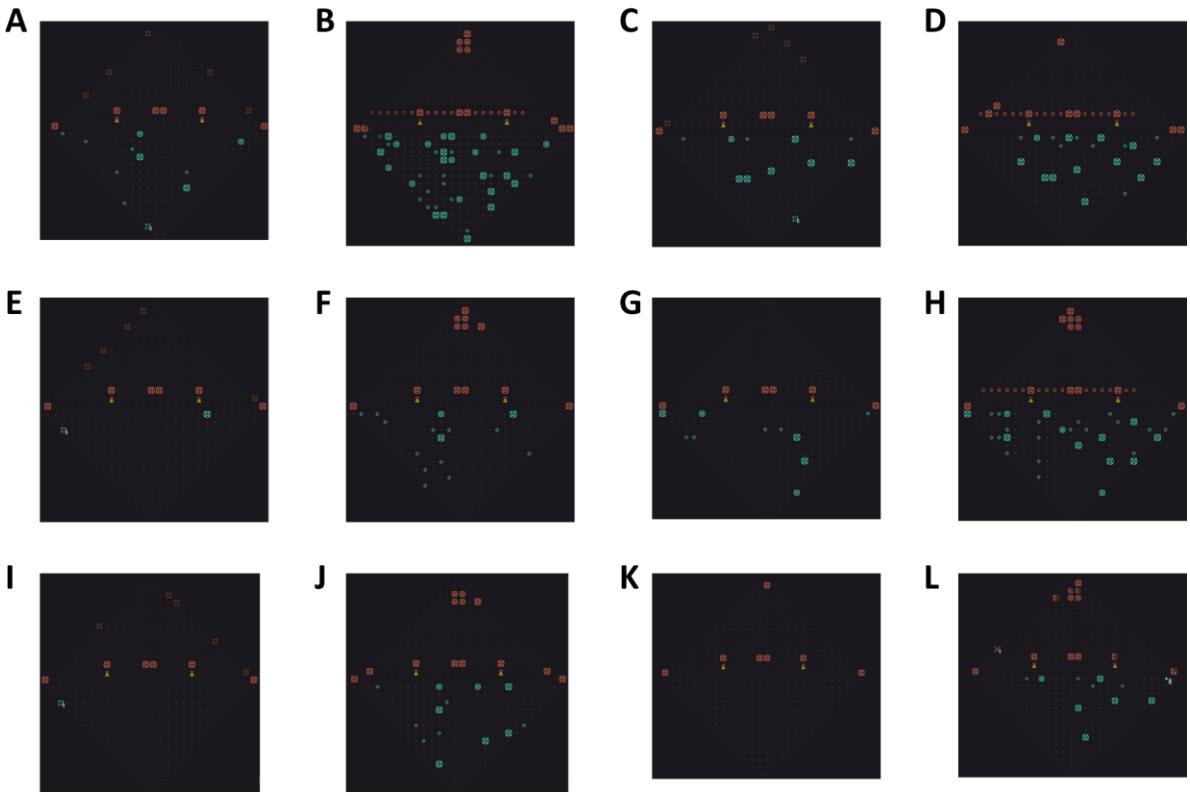

Figure 10. Gameboard of Terminal with agents with breached/scored locations

(A), (B) *BS_Python_100 (bottom) VS Python-algo (top) - start and end of game*
(C), (D) *BS_Copy_100_2 (bottom) VS Python-algo (top) - start and end of game*
(E), (F) *BS_Python_200 (bottom) VS Python-algo (top) - start and end of game*
(G), (H) *BS_Copy_200_1 (bottom) VS Python-algo (top) - start and end of game*
(I), (J) *BS_Python_300 (bottom) VS Python-algo (top) - start and end of game*
(K), (L) *BS_Copy_300_2 (bottom) VS Python-algo (top) - start and end of game*

Once again there is a large preference for Interceptors (**Table 4**). For agents trained on Python-algo, we do not observe a preference for Demolishers again, as seen in NoBS_Python_100. Looking at agents trained against a copy of itself, diversification of Mobile units increases as the number of training games increases. BS_Copy_100_2 solely uses Interceptors, while BS_Copy_200_1 starts using some Demolishers, and BS_Copy_300_2 begins using Scouts.

Units/Agent	BS_ Python_ 100	BS_ Python_ 200	BS_ Python_ 300	BS_ Copy_ 100_2	BS_ Copy_ 200_1	BS_ Copy_ 300_2
Scout	7.5%	3.9%	3.9%	0.0%	0.0%	3.4%
Demolisher	0.0%	0.0%	0.0%	0.0%	15.9%	12.8%
Interceptor	**92.5%**	**96.1%**	**96.1%**	**100%**	**84.1%**	**83.8%**

Table 4. Mobile unit deployment patterns of agents with breached/scored locations
Values are the percentage of Mobile points spent on the respective Mobile units, obtained from 1 winning game of the agent against Python-algo, where applicable. Scout and Interceptor cost 1 Mobile point, Demolisher costs 3 Mobile points.

4.3 Convolutional Neural Network

The agents here utilise a convolutional neural network. The conditions tested are as follows:

- 100 and 200 games against Python-algo - CNN_Python_100, CNN_Python_200
- 100 games against copy of itself (1) and (2) - CNN_Copy_100_1 and CNN_Copy_100_2
- 200 games against copy of itself (1) and (2) - CNN_Copy_200_1 and CNN_Copy_200_2

Surprisingly, we observe a trend of increase in rating from 100 training to 200 training games for both training conditions (**Table 5**). CNN_Copy_100_1 is left out due to its similar performance to CNN_Copy_100_2. CNN_Python increased from 1259 to 1451, while CNN_Copy increased from 645 to 1172 with an increase in training games. CNN_Python_200 has the highest rating in this batch, but it does not fare very well in terms of flexibility, losing four out of five games to Infiltrator, and losing all five games to Ironclad. Interestingly, CNN_Copy_200_1 is the first and only agent to lose any game to Foreman, losing two out of five games.

	Smite	CNN_ Python_ 100	CNN_ Python_ 200	CNN_ Copy_100 _2	CNN_ Copy_200 _1	CNN_ Copy_200 _2	Python-algo
Rating	1900	1259	1451	645	1081	1172	1366
Python- algo	-	0/5	5/0	1/4	0/5	0/5	-
Foreman	-	5/0	5/0	5/0	3/2	5/0	5/0
Infiltrator	-	2/3	1/4	0/5	0/5	1/4	1/4
Ironclad	-	1/4	0/5	0/5	0/5	0/5	0/5

Table 5. Results of agent performance without breached/scored locations and with CNN
Agents are assessed based on their rating, their performance against the baseline Python-algo, and their performance against different agents: Foreman, Infiltrator, and Ironclad. Performance is shown as a Win/Loss ratio in a series of five games.

Again, we do not observe the three-corner strategy being deployed by our agents (**Figure 11**). There is no obvious pattern of deployment from the agents, though it consistently disperses its units throughout the gameboard. We also observe that CNN_Copy_200_1 starts with a very sparse deployment of Structure units, and does not deploy anymore even as the game reaches its end (**Figure 11g, 11h**). This could be due to the exploitation of the reward function.

Using Interceptors seems to be a dominant strategy amongst all agents, but less so in this batch (**Table 6**). There is a significant diversification into using Scouts, most notably in CNN_Python_100, with 37.2% of Mobile points spent on Scouts. From 100 training games to 200 training games, it seems CNN_Python starts to value Interceptors more, and reduces its usage of Scouts. For CNN_Copy, the ratio of Scouts to Demolishers to Interceptors remains relatively constant throughout.

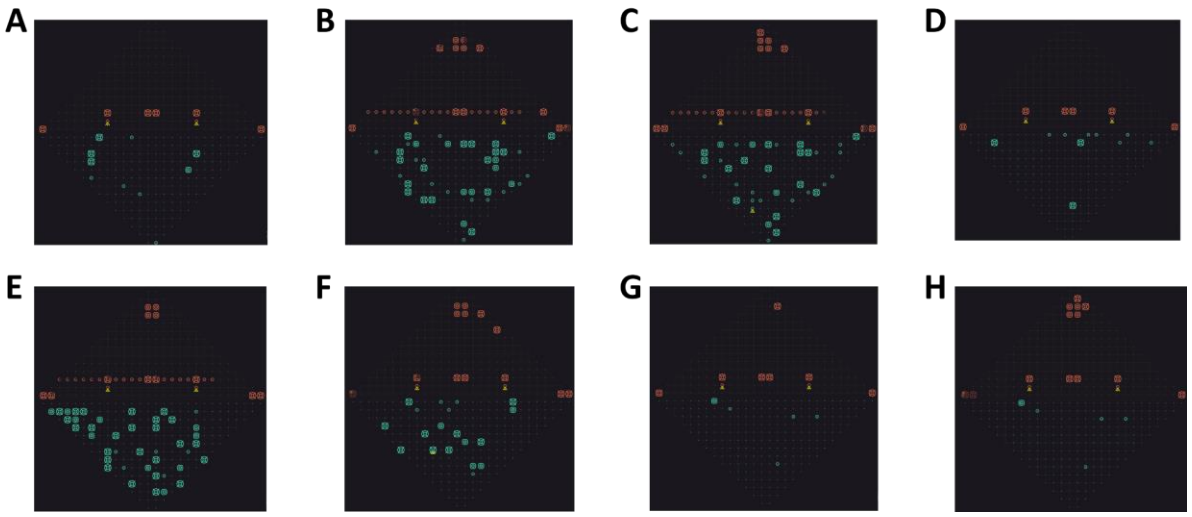

Figure 11. Gameboard of Terminal with agents without breached/scored locations, with CNN
(A), (B) *CNN_Python_100 (bottom) VS Python-algo (top) - start and end of game*
(C) *CNN_Python_200 (bottom) VS Python-algo (top) - end of game*
(D), (E) *CNN_Copy_100_2 (bottom) VS Python-algo (top) - start and end of game*
(F) *CNN_Copy_200_2 (bottom) VS Python-algo (top) - end of game*
(G), (H) *CNN_Copy_200_1 (bottom) VS Python-algo (top) - start and end of game*

Units/Agent	CNN_ Python_100	CNN_ Python_200	CNN_ Copy_100_2	CNN_ Copy_200_1	CNN_ Copy_200_2
Scout	37.2%	14.6%	15.1%	19.3%	16.2%
Demolisher	0.0%	1.6%	1.4%	1.1%	2.6%
Interceptor	**62.8%**	**83.8%**	**83.5%**	**79.6%**	**81.2%**

Table 6. Mobile unit deployment patterns of CNN agents
Values are the percentage of Mobile points spent on the respective Mobile units, obtained from 1 winning game of the agent against Python-algo, where applicable. Scout and Interceptor cost 1 Mobile point, Demolisher costs 3 Mobile points.

5. Discussion

Altogether, our group has demonstrated the ability of a Deep Q Network to learn on a large state and action space and came in at the top 1% of algos submitted. We optimised the action execution to only account for legal actions and tested out variations of the state space to cater to the agent's learning. We tested out different training conditions by varying the number of training games, and the opponent it trains against and found the best performing agent to be NoBS_Copy_100_2.

Rating, as a relative measure of performance, can fluctuate as the agent wins and loses against other algos on Terminal. It might not serve as a true measurement of an agent's performance, and so we supplemented it by testing its flexibility against Foreman, Infiltrator, and Ironclad. One interesting observation was that as the number of training games increases, the performance for agents trained against a copy of itself tends to flip between each copy. For example, in the BS_Copy batch, BS_Copy_100_2 has a higher rating and flexibility than BS_Copy_100_1. However, this reverses when the training games are increased to 200, with BS_Copy_200_1 outperforming BS_Copy_100_2. This again reverses when the training games are increased to 300. Likewise for CNN_Copy, and less obvious in NoBS_Copy. With an increase in the number of training games, a corresponding decrease in the number of Structure units deployed was also observed (**Figure 10, 11**). This is likely due to the reward function (**Figure 2**), where the agent exploits the difference in Structure units and points to maximise its reward, rather than trying to win the game and obtain the terminal reward.

We observe that the base agent performs well with a higher average rating across different training conditions. It is also the one that has an obvious strategy of placing units. A possible reason for the

breach and scored location agent and the CNN agent not performing as well could be because of the added complexity included in the state space which makes it more difficult for the TD learning to converge. Although the overall performance of the two agents is not that good, the performance for training against the python-algo is good. This shows that the DQN is working as intended to beat the python-algo in the game. Occam's razor may be at play here, whereby a simpler representation is better. These experiments, although costly, are useful for finding crucial game parts that are used in decision-making when trying to win a game.

In addition, END action is rarely used due to the large action space, which makes it difficult to learn the utility of this action. There is also limited diversity of attacks in the attack agent since deploying a unit changes the state by just one value, causing the DQN to output the same action in the next state. Furthermore, due to limited computation and memory resources, we had to reduce the size of replay memory size, learning batch, and action space. There are also limited algos to train against. Future work can be done by optimizing the neural networks with different layers and convolutions and refining the reward function to better reflect the path we want the agent to head towards. Different kinds of DQN agents can also be explored, utilizing better state and action space. Furthermore, rather than focusing on just DQN agents, hybrids of different artificial intelligence algorithms can be used.

6. Conclusion

In this project, we successfully created a Deep Q Network agent that placed in the top 1% of algos on the Terminal Season 8 competition. Our implementation can handle the large state and action space to give an agent good performance and flexibility against different strategies. Nevertheless, even our best performing agent, NoBS_Copy_100_2, is far from the very best within the competition. Improvements can be made to address the shortcomings described here, such as altering the reward function and optimisation of neural networks. Hence, our work here provides the foundation for the next generation of reinforcement learning algos on Terminal.

7. Distribution of labour

Tran Minh Duong → Final agent structure, action space, training, testing on website, report
Lim Feng Yue → Reward function, training, testing on website, presentation, report
Neo Jun Hao → Initial and final agent structure, state space, testing on website, presentation, report

8. Code and Demo

Github link - https://github.com/zwasd/terminal-agent

Games of NoBS_Copy_100_2 - https://youtube.com/playlist?list=PLay74ppqv5saVQyd26K-R1t4IwlkaI-sh

9. References

Brownlee, Jason. “Ordinal and One-Hot Encodings for Categorical Data.” *Machine Learning Mastery*, 11 June 2020, machinelearningmastery.com/one-hot-encoding-for-categorical-data/.

Correlation One. “Terminal.” *C1games.com*, 2021, terminal.c1games.com/. Accessed 20 Nov. 2021.

Draves, Ryan. “Ryan D’s Terminal Strategy.” *Terminal Player Strategies*, 20 Feb. 2019, medium.com/terminal-player-strategies/ryan-ds-terminal-strategy-eaa39f123a7e. Accessed 20 Nov. 2021.

Machine Learning with Phil. “Deep Q Learning Is Simple with Keras | Tutorial.” *Www.youtube.com*, 15 July 2019, www.youtube.com/watch?v=5fHngyN8Qhw&list=WL&index=1&t=1737s. Accessed 20 Nov. 2021.

“Terminal Forum.” *Terminal Forum*, forum.c1games.com/. Accessed 20 Nov. 2021.